PyTV: Python Templated Verilog

Wuqiong Zhao*

February 22, 2026

Abstract

PyTV (Python Templated Verilog)[1] is a Rust library and binary tool that extends Verilog with Python templating. By leveraging Python's flexibility, PyTV allows users to write Verilog templates with Python control constructs, including `if` statements, `for` loops, and function definitions. PyTV also supports instantiation of modules with complex parameters and enables hierarchy extraction.

Contents

1 Introduction **2**

2 Installation **3**
- 2.1 Prerequisites . . . 3
- 2.2 Install as a Rust Library . . . 3
- 2.3 Install as a Binary Tool . . . 3
- 2.4 Validate the Installation . . . 3
- 2.5 Build This Documentation . . . 3

3 Architecture **3**

4 Syntax Specifications **4**
- 4.1 Line Types and Markers . . . 4
- 4.2 Backtick Expression Substitution . . . 5
- 4.3 Indentation and Control Flow . . . 5
- 4.4 User Variables and Preamble . . . 6
- 4.5 Instantiation Blocks . . . 6
- 4.6 Dynamic Group Expansion in Instantiation . . . 6

*The author is with the University of California San Diego, La Jolla, CA, USA. e-mail: wqzhao@ucsd.edu.

[1] This is the documentation for PyTV version `0.5.6`. Latest digital version of this document is available at `https://go.wqzhao.org/pytv-docs`.

5 Usage **7**
5.1 Binary Tool Usage 7
5.2 Rust Library Usage 8
5.3 VS Code Extension 9

6 Limitations **9**
6.1 Python Execution Trust Model 9
6.2 Instantiation Scope 9
6.3 Verilog Parsing Model 9
6.4 Feature-Specific Behavior 9
6.5 License 10

7 Further Readings **10**

Acknowledgments **10**

References **10**

1 Introduction

Parameterized hardware generation often benefits from templating. PyTV follows this direction by combining Python as the control language and Verilog as the generated target. Compared with custom DSL-only approaches such as AHDW [1], PyTV intentionally keeps Verilog visible and editable in the source template.

The project provides two complementary interfaces:

- A Rust crate (`pytv`) for integration into toolchains and generators.
- A command-line binary (`pytv`) for direct file conversion.

At a high level, users write a `.pytv` file that mixes Verilog and Python directives. PyTV first converts it into a generated Python script, then optionally executes that script to emit:

- a rendered Verilog file (`.v`), and
- a module-instantiation metadata file (`.inst`) when instantiation blocks are used.

PyTV is open-source at `https://github.com/autohdw/pytv`, published on crates.io at `https://crates.io/crates/pytv`, and distributed under the GPL-3.0-or-later license.

2 Installation

2.1 Prerequisites

- Rust toolchain (`cargo`) for building or using the crate.
- Python 3 in `PATH` for rendering `.v` and `.inst` from generated Python.

PyTV invokes `python3` on Linux/macOS and `python` on Windows.

2.2 Install as a Rust Library

Add PyTV to an existing Rust project:

```
cargo add pytv
```

This updates `Cargo.toml` with a dependency entry similar to:

```
[dependencies]
pytv = "0.5.6"
```

Default crate features include `inst` and `macro`. If needed, enable or disable them explicitly in `Cargo.toml`.

2.3 Install as a Binary Tool

Install from crates.io:

```
cargo install pytv
```

Build from source:

```
cargo build --release
```

The compiled binary is available at `target/release/pytv`.

2.4 Validate the Installation

After installation, verify command-line availability:

```
pytv --help
```

2.5 Build This Documentation

From the `docs/` directory:

```
latexmk -pdf pytv-docs.tex
```

3 Architecture

The generation process of PyTV is illustrated in Figure 1.

The pipeline is intentionally split into two stages:

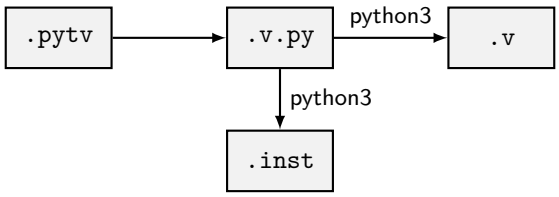


Figure 1: Architecture of PyTV.

- **Template lowering**: `.pytv` is converted to a Python script (`.v.py`).
- **Template execution**: the generated Python script is executed to emit target artifacts.

This separation has practical benefits:

- The generated `.v.py` is inspectable for debugging.
- Python runtime errors are surfaced with concrete script context.
- Users may choose to stop after stage 1 for auditing or for downstream processing.

By default, running PyTV without execution options generates only `.v.py`. When Python execution is enabled (CLI options `-r`/`-R`), PyTV writes:

- rendered Verilog to `.v`, and
- instantiation metadata to `.inst`.

Output naming follows these rules:

- If `-o`/`--output` is provided, that path is used as the Verilog output base.
- Otherwise, input extension is replaced by `.v` (or `.v` is appended if none exists).
- `.v.py` and `.inst` are derived from the chosen Verilog output base.

4 Syntax Specifications

4.1 Line Types and Markers

PyTV processes source code line by line. After leading whitespace is ignored, each line is classified as:

- **Inline Python line**: starts with `//<magic>` (default `//!`).
- **Python block line**: after a line starting with `/*<magic>` (default `/*!`), all following lines are Python until a line whose trimmed text is exactly `*/`. Text on the opener line itself is not emitted as Python code.

- **Verilog line**: any line not matched by the two rules above.

A minimal example:

```
//! a = 1 + 2
assign wire_`a` = wire_b;
/*!
b = a ** 2
*/
```

The magic marker is configurable by CLI option `-m/--magic`. Leading whitespace before markers is allowed.

4.2 Backtick Expression Substitution

In Verilog lines, content inside backticks (`` `...` ``) is treated as a Python expression and injected through Python f-string formatting in the generated `.v.py`. Matching uses a one-line regex equivalent to `` `([^`]+)` ``, so nested/multiline backtick expressions are not supported. For example:

```
//! w = 16
input [`w-1`:0] data_in;
```

renders as:

```
input [15:0] data_in;
```

PyTV escapes braces and single quotes while constructing the generated Python string, so literal `{` and `}` in Verilog text are preserved.

4.3 Indentation and Control Flow

Since Python controls generation, indentation semantics follow Python execution:

- Each input line has trailing whitespace removed.
- Tabs are converted to spaces using `-t/--tab-size` (default 4).
- For inline Python lines, the first non-empty inline Python line defines a baseline indentation.
- Subsequent inline Python lines must keep this baseline, otherwise conversion fails.
- In Python block mode, block contents are emitted as Python code (block delimiters are not emitted).
- Verilog lines are transformed into `print(...)` statements at the current Python indentation depth.

Practical consequence: any Python line ending with `:` (optionally followed by spaces or a comment) updates the indentation context for following generated Verilog print statements.

In inline mode, a common way to close a control-flow scope before more Verilog is:

```
//! for i in range(4):
foo_`i`
//! # return to baseline indentation
bar
```

4.4 User Variables and Preamble

CLI option `-v/--var KEY=VAL` injects user variables into generated Python before template processing. This is useful for build-system controlled feature flags and dimensions.

CLI option `-p/--preamble FILE` prepends a Python preamble file, allowing helper functions, imports, and shared constants.

4.5 Instantiation Blocks

Instantiation blocks are delimited by `<INST>` and `</INST>` inside Python regions. Their body uses YAML mapping syntax. The `<INST>` and `</INST>` markers must be standalone on their Python lines (except surrounding whitespace). A basic example:

```
//! <INST>
//! module: test_module
//! name: test_inst_`idx`
//! vparams:
//!   W: 32
//! ports:
//!   clk: clk
//!   rst_n: rst_n
//!   data: data_`idx`
//! </INST>
```

Field definitions are summarized in Table 1.

4.6 Dynamic Group Expansion in Instantiation

For `vparams` and `ports`, PyTV supports grouped expansion with keys prefixed by `!` (for example, `!auto:`):

```
//! dyn_ports = [("a", "sig_a"), ("b", "sig_b")]
//! <INST>
//! module: M
```

Table 1: Fields in an <INST> block.

Field	Required	Type	Meaning
module	Yes	scalar	Module name printed in Verilog and recorded in .inst.
name	Yes	scalar	Instance name printed in Verilog and recorded in .inst.
vparams	No	mapping	Verilog parameter map rendered as #(...) and recorded in .inst.
ports	No	mapping	Verilog port map rendered as (...) and recorded in .inst.
parameters	No	mapping	Metadata-only field stored in .inst; it is not rendered into Verilog instantiation text.

```
//! name: u0
//! ports:
//!   !auto: dyn_ports
//! </INST>
```

Here dyn_ports is a Python list of 2-tuples (name, value). At render time, PyTV expands the group into individual Verilog mappings and corresponding .inst entries. The group key after ! should use word characters ([A-Za-z0-9_]) for predictable parsing. <INST> blocks cannot be nested.

5 Usage

5.1 Binary Tool Usage

Quick Start

Generate only the intermediate Python script (.v.py):

```
pytv path/to/design.pytv
```

Generate Verilog/.inst and keep .v.py:

```
pytv path/to/design.pytv --run-py
```

Generate Verilog/.inst and delete .v.py:

```
pytv path/to/design.pytv --run-py-del
```

Set an explicit output base file:

```
pytv path/to/design.pytv -o build/design.v -R
```

Examples in This Repository

```
cargo run -- examples/test.pytv -R
cargo run -- examples/D1.pytv -R \
```

```
--preamble examples/D1_preamble.py
```

Command-Line Options

Command-line options are summarized in Table 2.

Table 2: PyTV command-line options.

Option	Description
`INPUT`	Input `.pytv` file (required positional argument).
`-o, --output FILE`	Set output Verilog path/base. Derived `.v.py` and `.inst` names follow this base.
`-r, --run-py`	Execute generated Python and keep `.v.py`.
`-R, --run-py-del`	Execute generated Python and delete `.v.py` after success.
`-t, --tab-size INT`	Tab width used when normalizing tab characters in input lines (default: 4).
`-m, --magic STRING`	Magic marker string used after `//` and `/*` (default: `!`).
`-v, --var KEY=VAL`	Inject Python variables; option can be repeated.
`-p, --preamble FILE`	Prepend a Python preamble script before template conversion.

5.2 Rust Library Usage

PyTV can be integrated directly in Rust applications. The core types are `Config`, `FileOptions`, and `Convert`.

Programmatic Conversion Example

```
use pytv::{Config, Convert, FileOptions};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let config = Config::new(
        "!".to_string(),
        Config::default_template_re(),
        true,  // run generated Python
        true,  // delete .v.py after successful run
        4,     // tab size
    );

    let files = FileOptions {
        input: "examples/test.pytv".to_string(),
        output: Some("build/test.v".to_string()),
    };

    let vars = Some(vec![
        ("if_rst".to_string(), "True".to_string()),
```

```
        ("if_en".to_string(), "False".to_string()),
    ]);

    let convert = Convert::new(config, files, vars, None);
    convert.convert_to_file()?;
    Ok(())
}
```

5.3 VS Code Extension

A VS Code extension with syntax highlighting is available for PyTV. Install by searching for `pytv` in the VS Code Marketplace, or visit the VS Code Marketplace page.

6 Limitations

Important: *PyTV is intended for research, education, and rapid prototyping workflows. Production deployment requires additional validation and process controls.*

6.1 Python Execution Trust Model

PyTV executes generated Python when `--run-py` or `--run-py-del` is used. Any Python embedded in `.pytv` or preamble files runs with the invoking user's permissions. Treat templates and preambles as trusted code.

6.2 Instantiation Scope

`<INST>` blocks support single-level instantiation extraction per module conversion. Nested `<INST>` blocks are rejected. Hierarchical assembly across modules should be implemented by a higher-level orchestrator using emitted `.inst` files.

6.3 Verilog Parsing Model

PyTV is line-oriented and template-oriented. It does not perform full HDL semantic checks. Backtick substitution and emitted text correctness remain the user's responsibility.

6.4 Feature-Specific Behavior

- With feature `inst` disabled, instantiation parsing/output is unavailable.
- The optional `parameters` field in an `<INST>` block is stored in `.inst` metadata but is not rendered into Verilog module instantiation text.

6.5 License

PyTV is distributed under GPL-3.0-or-later. Downstream usage must comply with the GPL terms.

7 Further Readings

Hardware generation with software programming languages is an active area of exploration. AHDW [1] is an earlier implementation attempt in this direction. FLAMES, a C++ high-level synthesis (HLS) library for Vitis HLS, was proposed in [2].

Acknowledgments

The author would like to thank members of the `@autohdw` team for their support throughout the development of this project. The author would also like to thank *GitHub Copilot* and *ChatGPT Codex* for code suggestions during development and for assistance in drafting this document.

References

[1] W. Zhao, C. Li, Z. Ji, Y. You, X. You, and C. Zhang, "Automatic timing-driven top-level hardware design for digital signal processing," in *Proc. IEEE Int. Conf. ASIC (ASICON)*, Nanjing, China, Oct. 2023, pp. 1–4.

[2] W. Zhao *et al.*, "Flexible high-level synthesis library for linear transformations," *IEEE Trans. Circuits Syst. II*, vol. 71, no. 7, pp. 3348–3352, Jul. 2024.